Rapport de projet

Environnement de Programmation

BONNET Thomas , JAJOUX Jeremy

1 Fonctions de base du jeu

Les fonctions de grid.c

Ici la plupart des fonctions étaient simples à programmer car elles sont de constructions moindres et ont un objectif unique et court.

Les fonctions ayant le plus posé de problèmes furent les fonctions relatives aux mouvements des tuiles, c'est à dire "can_move" et "do_move", car ces dernières nécessitent de prendre en compte toutes les différentes situations possibles lors d'un mouvement ou d'une fusion de tuiles.

C'est pourquoi, lors des premières ébauches du code, toutes ces fonctions étaient très longues et répétitives afin d'inclure toutes les possibilitées sans se soucier de la structure de celles-ci.

L'étape suivante a donc consisté à réduire la quantité de code et à créer des fonctions annexes, comme "move" ou "tile_fusion", pour alléger le code de chaque fonction et permettre une élimination des répétitions avec des sous-fonctions dotées de différents arguments pour fonctionner selon la situation plutôt qu'une section de code répétée avec de légères modifications.

Par exemple dans la sous-fonction suivante, les variables x et y qui parcourent la grille peuvent être interchangées afin de parcourir la grille de bas en haut et de droite à gauche. Les variables i et j repèrent la cellule de destination et les variables ii et jj la cellule vide suivante dans le cas où un mouvement est possible mais pas une fusion.

```
static int
move(grid g,int x,int y,int i,int j,int ii,int jj){
  int a=0;
  if(get_tile(g,x,y)!=0){
    if(get_tile(g,i,j)==0)
      move_tile(g,x,y,i,j);
    else{
      if(get_tile(g,x,y)==get_tile(g,i,j))
tile_fusion(g,x,y,i,j);
      else
move_tile(g,x,y,ii,jj);
      a=1;}
  }
  return a;
}
```

On arrive alors à un code relativement compact et malheureusement un peu illisible si on ne l'a pas écrit.

Un autre obstacle s'est manifesté dans la fonction "add_tile", cette dernière nécessite de prendre en compte toutes les cellules vides et ajouter de manière aléatoire dans une de ces cellules la valeur 2 ou, dans 10% des cas, la valeur 4.

Pour ce faire il a fallu créer un tableau de cellules vides en parcourant la grille, puis à l'aide de la fonction "random" séléctionner une des cases du tableau correpondant à une cellule vide au hasard. On ajoute ensuite une tuile de valeur décidée également à l'aide de random. L'utilisation de random s'est faite à tâton, plus particulièrement l'initialisation de la "seed" en tandem avec la fonction "time" qui devait se trouver avant l'éxecution du jeu dans son entièreté.

Au final ce fut une partie du projet qui a nécessité beaucoup de temps et de changement pour fonctionner comme désiré et de manière la plus optimisée possible.

2 Fonctions pour faire fonctionner le jeu

Les fonctions contenues dans play.c

Cette partie du code a été relativement compliquée à faire fonctionner et a nécessité un grand nombre de changement que ce soit pour améliorer son fonctionnement, optimiser son code ou ajouter de nouvelles options.

Plus que la création et l'affichage de la grille qui sont d'une écriture simple, le réel problème fut de faire fonctionner la grille comme voulu en entrant une touche. Plusieurs problèmes se sont accumulés pour ce problème précis, tout d'abord il fallait réussir à lire une touche entrée par l'utilisateur, ceci a été accompli à l'aide de la fonction "getchar" qui, couplée à la fonction "getDirection" pour sélectionner la direction en fonction de la touche pressée, a permis de faire fonctionner le jeu correctement. Cependant "getchar" renvoie l'un après l'autre la touche pressée et la touche Entrée utilisée pour la validation, ce qui faisait faire deux fois le même mouvement à la grille. Nous avons pallié à cela en ajoutant à l'exécution du programme une variable booléenne "change" qui permettait de vérifier qu'une touche de direction valide venait d'être pressée et ne l'exécute qu'une fois. Ce problème n'est étonnamment pas apparu dans la version ncurses du jeu, probablement dû à un fonctionnement différent de "getch".

Un autre problème, qui a été plus rapide à régler, était de faire en sorte que le programme termine bien le jeu. Pour cela il a fallu à nouveau ajouter une variable boléenne "playing" qui voit sa valeur changer en accord avec la fonction "game_over" après chaque exécution de mouvement et stoppe la fonction lorsque plus aucun mouvement n'est possible.

```
void
play_terminal(grid g){
  bool playing=true;
  bool change=true;
  int c;
  dir d;

  afficher_grille_terminal(g);
  while(playing){
```

```
    c=getchar();

    if(validDirection(c)){
      d = getDirection(c);
      change=true;
    }

    if(c == TX)
      playing = false;

    else if(can_move(g,d) && change){
      play(g,d);
      afficher_grille_terminal(g);
      playing=!game_over(g);
    }
    change=false;
  }
}
```

Le programme est également doté d'une partie après le jeu qui permet d'afficher à la fin du jeu le score obtenu et la plus grande valeur de tuile obtenue dans le but de faciliter de futurs tests d'efficacité sur les stratégies.

3 Les Tests

Trouvés dans test.c

L'objectif de cette partie était de tester le bon fonctionnement des fonctions, en remplissant des grilles de façon à créer des situations et vérifier que le résultat obtenu est bien le résultat attendu dans cette situation. L'aspect le plus important de cette partie était donc de faire en sorte de couvrir toute les situations possibles. Cette reflexion s'applique principalement pour les fonctions "can_move" et "do_move" qui peuvent chacune être utilisées dans 4 directions différentes. Les tests des autres fonctions restent relativement simples, s'agissant dans la plupart des cas d'une comparaison de grilles.

Dans le cas de la fonction "can_move", il faut vérifier que la fonction ne renvoie pas tout le temps true ou false mais aussi qu'elle renvoie bien true

lorsqu'un mouvement est possible et false lorsqu'un mouvement n'est pas possible. Pour cela une grille est remplie manuellement de façon à n'autoriser que deux mouvement. Le test consiste donc a vérifier le résultat de la fonction "can_move" sur la grille précédemment remplie.

Le test de la fonction "do_move" est sensiblement le même. La grille utilisée pour ce test est la même que celle utilisée pour le test "can_move" car en plus de n'autoriser que certaines directions, elle ne permet aussi aucune fusion. Ainsi lorsqu'un mouvement est réalisé dans une direction possible, il n'est plus possible de refaire un mouvement dans cette même direction. Le test consiste donc pour cette fonction à vérifier qu'un mouvement dans une direction est impossible après avoir effectué un mouvement dans cette même direction.

4 Le Makefile

Il y a peu de chose à dire ici car nous n'avons pas beaucoup d'expérience dans la création d'un Make fonctionnel, cependant il permet de compiler le programme du jeu, les tests et la librairie sans problème.

Il est regrettable que nous n'ayons pas fait en sorte que tout soit disposé dans différents dossiers et la compilation doit se faire à la racine où se trouvent tous les fichiers.

5 Les Stratégies

Probablement la partie la plus infructueuse du projet. La stratégie que l'on a ici a été créée à partir d'un concept simple et contre toute attente marche correctement, les stratégies rapides et efficientes sont les mêmes avec seulement une différence de profondeur.

La stratégie part du fait que le score est changé lors d'une fusion et augmente de la valeur de la nouvelle tuile. Ainsi on peut supposer que si le score est grand, on possède également des tuiles de valeur importante, puisqu'on veut atteindre la tuile 2048 c'est intéressant. Cette stratégie exécute donc une simple récursion en jouant les quatre directions possibles sur autant de

copies de la grille actuelle et conserve la direction qui aboutit au meilleur score. On ajoute ici une varible “depth” qui répète cette opération pour autant de mouvements à la suite qu'indiqué par sa valeur.

Ceci ne devrait pas très bien marcher à cause la nature aléatoire du placement de tuile, ainsi un mouvement prédit par la fonction obtiendra le même score que le mouvement réel mais tout mouvement subséquent provoqué par la récursion n'aura pas nécessairement la même valeur que le mouvement réel. En effet la version efficiente de la stratégie qui possède une valeur de depth plus élevée est moins efficace que la version rapide, ceci vient du fait qu'en augmentant le nombre de récursion on augmente le nombre de fausses prédictions.

Cependant faute de meilleure stratégie c'est celle-ci que nous conservons.

6 Interface Graphique

Le corps du fichier 2048.c

Le but de cette partie était de pouvoir jouer au jeu 2048 sur une interface graphique en dehors du terminal. Pour cela l'interface graphique de l'application a été réalisée à l'aide de la bibliothèque MLV.

L'affichage consiste principalement en un positionnement de “text box” carrés, représentant chacun une tuile de la grille, de même taille les uns à la suite des autres, en modifiant les coordonnées de ces derniers après chaque placement, dans le but de créer une grille de taille GRID_SIDE.

Chacun de ces carrés contient ainsi la valeur correspondant à son emplacement dans la grille passée en paramètre. Dans un souci de lisibilité, ces carrés sont de couleurs différentes en fonction de la couleur qu'ils contiennent. Pour cela, une fonction “pickColor” a été mise en place, renvoyant la couleur correspondant à la valeur de la tuile passée en paramètre. Cette fonction est donc appelée avant la création de chacun des textbox pour récupérer la couleur correspondant à la valeur qu'il contient.

En dehors de la grille, on retrouve uniquement le score positionné en

dessous de la grille ainsi qu'un court message indiquant les commandes de jeu.

L'un des problèmes rencontré lors de la création de cet affichage, est lié à la façon dont la bibliothèque MLV nous permet de gérer la police d'écriture. En effet, il n'est pas possible de choisir ne serait-ce que la taille de la police lors de la création d'un text box ou d'un texte.
Pour cela, il est nécessaire de charger une police d'écriture dans une variable de type MLV_Font, en spécifiant le chemin de l'écriture et sa taille, et de la passer en paramètre lors de la création du text box ou du texte pour pouvoir l'utiliser.

```
MLV_Font* font_footer = MLV_load_font( "fonts/helveticaneue.ttf" , 20 );
  MLV_draw_text_box_with_font(0, HAUTEUR - 100, LARGEUR, 100,
   "Play with ZQSD || 8456 || Arrows keys\n Press X to QUIT.",
    font_footer, 6,COLOR_BACKGROUND, COLOR_TEXT, COLOR_BACKGROUND,
    MLV_TEXT_CENTER,MLV_HORIZONTAL_CENTER, MLV_VERTICAL_CENTER);

MLV_free_font(font_footer);
```

Il est ensuite nécessaire de libérer la police chargée en faisant un MLV_free_font après chaque affichage de la grille, l'application n'arrivant plus à charger la police après un certain temps d'exécution sans cette instruction, faisant ainsi planter l'application.

L'un des autres points importants de cette partie était l'affichage de messages en fin de partie, que ce soit une victoire ou un game over.
Ainsi lorsqu'une fin de partie est détectée, un message est affiché par dessus la grille indiquant à l'utilisateur qu'il a perdu en cas de game over, ou qu'il a gagné en cas de victoire. Cependant nous avons fait le choix de pouvoir laisser la liberté à l'utilisateur de pouvoir continuer de jouer même après avoir gagné (avoir atteint 2048). Pour résoudre ce problème, un booléen est vérifié avant l'appel de l'affichage pour déterminer si la victoire doit être affichée ou non.
Concrètement, lors de la premiere victoire, le booléen rentrera dans la condition et la victoire sera ainsi affichée, proposant à l'utilisateur de continuer de jouer. Ensuite si le joueur choisit de continuer de jouer, l'état du booléen sera modifié, bloquant l'affichage de la victoire pour les futurs mouvements.